大規模最短路問題に対するダイクストラ法の高速化 (最適化モデルとアルゴリズムの新展開)

AUTHOR(S):

安井, 雄一郎; 藤澤, 克樹; 鳥海, 重喜; 田口, 東

CITATION:

安井, 雄一郎 ...[et al]. 大規模最短路問題に対するダイクストラ法の高速化 (最適化モデルとアルゴリズムの新展開). 数理解析研究所講究録 2011, 1726: 62-72

ISSUE DATE:

2011-02

URL:

http://hdl.handle.net/2433/170508

数理解析研究所講究録
第1726巻 2011年 62-72

大規模最短路問題に対するダイクストラ法の高速化

中央大学理工学研究科経営システム工学専攻　安井雄一郎 (Yuichiro Yasui)
Department of Industrial and Systems Engineering, Chuo University
中央大学理工学部経営システム工学科　藤澤克樹 (Katsuki Fujisawa)
Department of Industrial and Systems Engineering, Chuo University
中央大学理工学部情報工学科　鳥海重喜 (Shigeki Toriumi)
Department of Information and System Engineering, Chuo University
中央大学理工学部情報工学科　田口東 (Azuma Taguchi)
Department of Information and System Engineering, Chuo University

概要

最短路問題はネットワーク上の経路探索や他の最適化問題の小問題となるなどの適用範囲の広い組合せ最適化問題であり, 高速に解くことの重要性は非常に大きい. 最短路問題に対する解法にはダイクストラ法などの安定的かつ効率的な高速アルゴリズムが存在するものの, 実問題は非常に大規模であるため高速化が不可欠である. そこで本論文では大規模最短路問題に対して計算機のメモリ階層構造を考慮した高速化方法を示していく. 本手法で実装されたバイナリヒープを適用したダイクストラ法は, 特定の問題サイズや問題特性に限定することなく汎用的に高速化されており, 現在一般的なマルチコアプロセッサ計算機環境上で非常に効率的に実行することが可能である. 他の実装と比較して, 実行性能, 安定性, メモリ要求量など総合的に最も優れているといえる.

1 はじめに

最短路問題は適用範囲の広い組合せ最適化問題であり, ネットワーク上の経路探索などの多くの応用を持ち, 他の最適化問題の子問題として用いられることも多い. 最短路問題を高速に解くことの重要性は非常に大きいため, アルゴリズムに対し理論的な計算量の見積もりをするだけでなく, 実装後の性能も活発に議論されている [5]. 最短路問題に対する解法にはダイクストラ法 [1] などの安定的かつ効率的な高速アルゴリズムが存在するが, 実社会から生成される最短路問題は非常に大規模であるため高速化が不可欠である. 本研究で扱う時空間ネットワークに変換した首都圏鉄道ネットワークグラフ (約78万点, 約311万枝)[15, 16, 17], 全米道路ネットワークグラフ (約2400万点, 約5800万枝)[7] も同様に大規模であるため, 優先キュー付きダイクストラ法 [2, 3] に対し, 計算機のメモリ階層構造を考慮し高速化 [10, 11, 12] を行った. 本手法で実装されたバイナリヒープを適用したダイクストラ法は, 特定の問題サイズ, 問題特性限定することなく汎用的に高速化されており, 他の実装と比べ実行性能, 安定性, メモリ要求量など総合的に最も優れているといえる. 論文中では計算機のメモリ階層構造上でのボトルネック箇所の特定を行うための実験方法も合わせて示し, 本実装の性能を検証していく [18].

2 高性能最短路問題ソルバ

2.1 最短路問題

各枝の重みが非負整数である有向グラフ $G=(V,E,l), l:(v,w)\to \mathbf{Z}^+((v,w)\in E)$ に対する1対1最短路問題 (Point-to-Point Shortest Path Problem; P2PSP), 1対全最短路問題 (Single-Source Shortest Path Problem; SSSP) について表1にまとめる. 問題間には P2PSP $\subset$ SSSP という包含関係が成り立つため, SSSP の出力から同一始点の P2PSP の出力を得ることが可能である.

最短路問題を効率的に計算するためのアルゴリズムとしてダイクストラ法が有名であり, $O(n^2)$(n は点数)で SSSP を計算することができる [1]. ダイクストラ法は本来 SSSP に対するアルゴリズムであるが, P2PSP も効率的に計算可能である. ダイクストラ法のボトルネック箇所である次探索点候補の取り扱いに優先キュー

表 1: 最短路問題の種類

	1 対 1 最短路問題 (P2PSP)	1 対全最短路問題 (SSSP)
入力	有向グラフ $G(V,E,l)$ (始点 s, 終点 t), $s,t \in V$	有向グラフ $G(V,E,l)$ (始点 s), $s \in V$
出力	(s,t)-最短路 (s,t)-最短路長	s を根とする最短路木 ((s,v)-最短路, $\forall v \in V$) 最短路木長 ((s,v)-最短路長, $\forall v \in V$)

と呼ばれるデータ構造を適用することで, バイナリヒープ [2] では $O((n+m)\log n)$, 1 レベルバケット [3] では $O(m+nU)$, マルチレベルバケット [6] では $O(m+n\log U)$(n は点数, m は枝数, U は最大枝長) と改善される.

2.2 ダイクストラ法に対する優先キューの適用

ダイクストラ法に対する優先キューは, `insert`, `decrease-key`, `extract-min` という操作に対応したデータ構造である (表 2 参照). 優先キュー付きダイクストラ法の実行には, ダイクストラ法と同様に各点 $v \in V$ に対し, 始点からの距離ラベル $d(v)$, 最短路における直前の訪問点 (確定されていない場合では `nil` とする) $\pi(v)$ が必要となる (Algorithm1).

優先キュー付きダイクストラ法の終了条件は, SSSP では優先キューが空になること (Algorithm1:5 行目), P2PSP では終点 t が探索点になること (Algorithm1:6 行目で得た v が終点である) である. SSSP, P2PSP いずれにおいても, 各点は高々1 回だけ探索点となり, 各枝も高々1 回だけ探索される.

表 2: 優先キューの操作

	説明
`insert`	点 $v \in V$ を, 距離ラベル $d(v)$ を優先度として優先キュー Q に挿入する
`decrease-key`	点 $v \in Q$ に対して, 距離ラベル $d(v)$ を $d'(v)$ ($d'(v) < d(v)$) に更新する
`extract-min`	距離ラベル $d(v)$ が最小の点 $v \in Q$ を取り出す

Algorithm 1 SSSP(1 対全最短路問題) に対する優先キュー付きダイクストラ法

Require: グラフ表現 $G=(V,E,l)$, 優先キュー $Q=\emptyset$, 始点 $s \in V$
Ensure: 始点からの各点の距離ラベル d, 最短路における直前点 π

```
1: d[v] ← ∞, v ∈ V
2: π[v] ← nil, v ∈ V
3: d[s] ← 0
4: insert(Q, s)
5: while Q ≠ ∅ do
6:    v ← extractmin(Q)
7:    for all w : (v, w) ∈ E do
8:       if d[w] > d[v] + l(v, w) then
9:          d[w] ← d[v] + l(v, w)
10:         if π[w] = ∅ then
11:            insert(Q, w)
12:         else
13:            decreasekey(Q, w)
14:         end if
15:         π[w] ← v
16:      end if
17:   end for
18: end while
19: return d, π
```

2.3 優先キューの種類と特性

ダイクストラ法は適用した優先キューの特性に依存することになるため, 優先キューの選択は注意して行わなければならない. 表 3 は, 全米道路ネットワークグラフ (点数 $n = 23,947,347$, 枝数 $m = 58,333,344$, 最大枝長 $U = 368,855$) に対する SSSP に対し, A.V.Goldberg により実装された優先キュー付きダイクストラ法 [5, 6] の実行時間をまとめたものである. 優先キューなしのダイクストラ法 (DIKQ) に対して, 優先キューを適用することで大きな性能向上が確認されるものの, フィボナッチヒープ [4] を適用したダイクストラ法 (DIKF) のように計算量から期待される性能が得られないといった現象も確認される. また, 1 レベルバケット, ダブルバケットを適用したダイクストラ法 (DIKB, DIKBD) はグラフ特性に依存しやすいため, 計算量のみの比較だけでは実装後の性能を予想することは非常に困難である.

表 3: 優先キュー付きダイクストラ法の性能 [5, 6]

	計算量	アルゴリズム	SSSP[sec.]	Memory [GB]
DIKQ	$O(n^2)$	naive Dijkstra's algorithm	674.13	1.81
DIKH	$O(m \log_k n)$	k-ary heap ($k = 4$)	7.23	2.43
DIKR	$O(m + n \log U)$	1-level R-heap	8.09	2.80
DIKF	$O(m + n \log n)$	fibonacci heap	15.97	3.17
DIKB	$O(m + nU)$	Dial's algorithm	4.38	2.62
DIKBD	$O(m + n(\Delta + U/\Delta))$	double buckets $\Delta = \lceil C/2^{11} \rceil$	4.67	2.62
mbp	$O(m + n \log U)$	multi-level buckets	5.68	2.17

さらに, DIKH, DIKB, mbp[6] を用いて優先キューの典型的な特性をまとめる (図 1, 2 参照). 用いたグラフは 9thDIMACS[7, 8, 9] のベンチマーク問題である Random4-C であり, 点数・枝数を 1,048,576 点・4,194,304 枝と固定し, 図中の横軸パラメータの i により最大枝長 4^i を決定している. なお, 計算機環境は Harpertown(表 6 参照) である. いずれのグラフにおいても DIKB は実行時間やメモリ要求量が最大枝長に依存することが確認される. 一方 DIKH と mbp は最大枝長に対する依存度が低く安定的である.

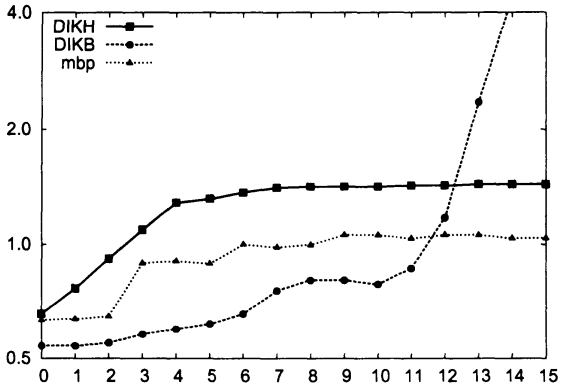


図 1: Random4-C.i.0.gr: 実行時間 [sec.]

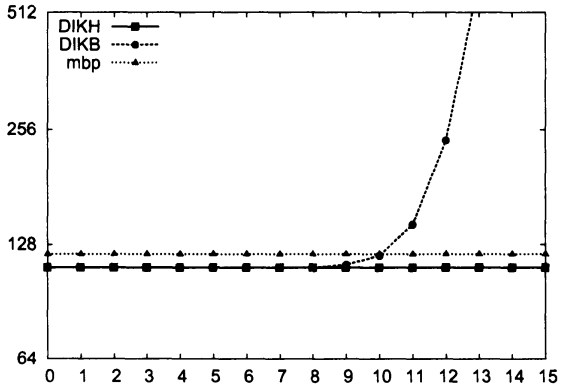


図 2: Random4-C.i.0.gr: メモリ要求量 [MB]

本論文では, ヒープを用いた優先キューとバケットを用いた優先キューの, それぞれ最も基本的な優先キューであるバイナリヒープと 1 レベルバケットを適用したダイクストラ法に対し, 計算機のメモリ階層構造を考慮した高速化を行い, 9th DIMACS での参照実装である mbp と比較していく.

2.4 メモリ階層構造を考慮した高速化

メモリ階層構造を考慮した高速化には特に次のような点に注意している.C 言語で実装した.

1. 特定のアーキテクチャや問題特性に特化させることなく汎用的に高速化を行うこと
2. 最近のマルチコア・プロセッサを搭載した計算機環境を想定すること

表 4: 優先キューの種類

	ヒープを用いた優先キュー	バケットを用いた優先キュー	
優先キュー	バイナリヒープ	1 レベルバケット	マルチレベルバケット
構成方法	木構造 (2 分木)	バケット&双方向リスト	バケット&双方向リスト
優先度の分類方法	大小比較 (スワップ)	最大枝長 $U+1$(余算)	2 の冪乗 (ビット操作)
性能と特性	安定的	最大枝長 U に依存	安定的
`insert()` の計算量	$O(\log n)$	$O(1)$	$O(1)$
`decreasekey()` の計算量	$O(\log n)$	$O(1)$	$O(1)$
`extractmin()` の計算量	$O(\log n)$	$O(U)$	$O(\log U)$
SSSP の計算量	$O((n+m)\log n)$	$O(m+nU)$	$O(m+n\log U)$

性能効率を出しやすい条件とダイクストラ法の特性は次のようにまとめられる. これらからダイクストラ法は高速化が容易ではないアルゴリズムといえる.

1. データ移動量に対し計算量がある程度大きいこと (つまり $\frac{\text{計算量}}{\text{データ移動量}}$ の比がある程度大きいこと)
   → ダイクストラ法では, データ移動量に対し演算量が非常に小さい
2. データアクセスが連続的で, 中長期的に予測が可能であること
   → ダイクストラ法では, 不連続な領域に対しデータアクセスが広域に及ぶため中長期的な予測が困難

2.4.1 メモリ階層構造

論文中では計算機内部が図 3 のようなメモリ階層構造になっていると仮定する [13]. メモリ階層構造では, 上位レベルになるほどアクセス速度が高速で小容量な記憶領域を, 下位レベルになるほどアクセス速度が低速で大容量な記憶領域を保持している. 特に CPU 内部では, レジスタやキャッシュメモリ, TLB [1]など非常に高速な記憶領域が存在する. 演算処理はレジスタ上でのみ行うことができるが, 非常にアクセス速度が高速 (250ps) かつ容量が非常に小さい. また主記憶装置 (RAM) は数 Gbytes 以上と非常に大容量であるがアクセス速度はレジスタと比較すると極めて低速である (100ns). そのため最適化分野に限らずソフトウェアの高速化のためには, 演算量とデータ移動量の割合を考慮してデータを適切に配置し, レジスタと主記憶装置の中間に位置するキャッシュメモリを有効に利用することは非常に重要である. L2 (あるいは L3) キャッシュメモリは, 比較的高速で数 Mbytes という大きな容量を保有しており, かなり大きな 1 次元配列でも格納することができる. そのため後述するようにダイクストラ法の高速化においては特に重要性が高い.

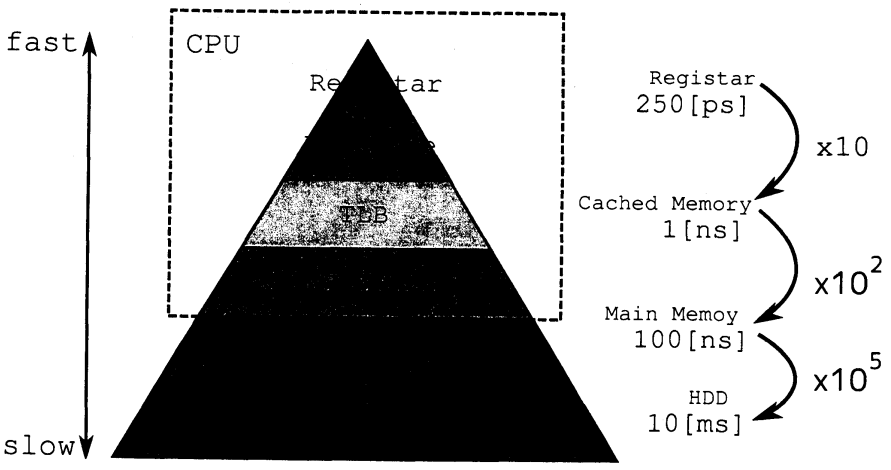


図 3: メモリ階層構造の例

[1]仮想メモリアドレスと物理メモリアドレスの変換のためのバッファメモリ

2.4.2 データアクセスパターンを考慮したデータ配置

ダイクストラ法は, 非常に広域なデータに対して中長期的な予想が困難な再利用率の低いデータアクセスを繰り返し行うため, アクセスパターンを考慮したデータ配置を行うことが非常に重要である.

点数 n, 枝数 m のグラフを表現するために要求される領域は, 隣接行列表現 (adjacency-matrix represen-taion) では $O(n^2)$, 隣接リスト表現 (adjacency-list representaion) では $O(n+2m)$ となる [14]. 道路ネットワークのように疎性の高いグラフに対し隣接行列表現を適用すると, 不連続なデータアクセスとなってしまう (図 4 参照). 配列を用いて実装した隣接リスト表現に対してアクセスパターンを考慮したデータ配置を適用することで, 局所的ではあるが連続的なデータアクセスに改善することが可能である (図 5 参照).

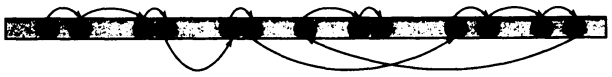
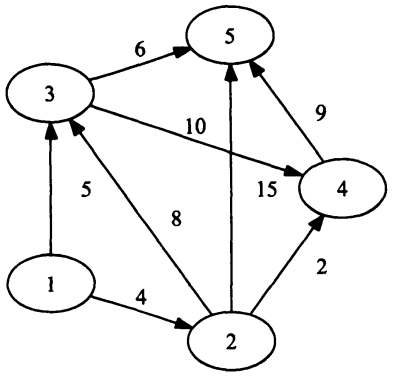


図 4: 不連続なデータアクセスパターン

図 5: 局所的に連続なデータアクセスパターン

ここで, 隣接リスト表現の枝情報配列 `arc[]` のデータ配置方法として, 終点・枝の重みを要素とする構造体の配列を用いた実装 A_{struct} (図 7 参照) と, 終点・枝の重みをそれぞれ個別の配列 `head[]`, `length[]` とした実装 A_{ary} (図 8 参照) の 2 種類が考えられる. また, ダイクストラ法の作業領域となる各点に対する作業領域 (距離ラベル d, 直前訪問点 π) や, 優先キューに対しても, 同様に 2 種類の配置方法 N_{struct}, N_{ary} が考えられる.



incident[]

NodeID	1	2	3	4	5	(6)
ArcID	0	2	5	7	8	8

arc[]

ArcID	0	1	2	3	4	5	6	7
head	2	3	3	4	5	4	5	5
length	4	5	8	2	15	10	6	9

図 6: 配列により実装された隣接リスト表現の例 (点数 5, 枝数 8)

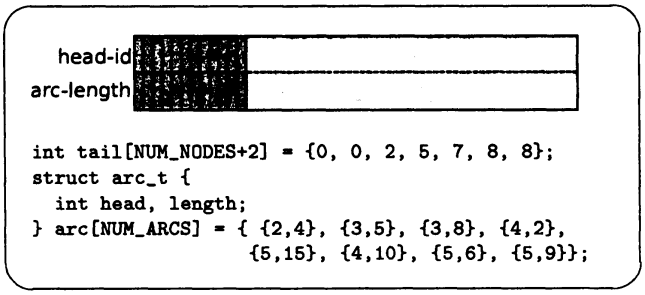


図 7: 構造体の配列による実装 A_{struct}

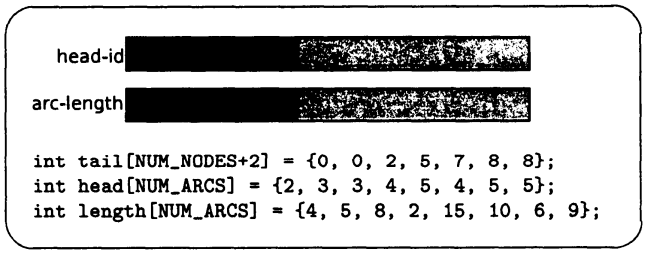


図 8: 個別の配列による実装 A_{ary}

図 9, 表 5 は, グラフ表現に対するデータ配置方法 A_{ary}, A_{struct}, ダイクストラ法の作業領域と優先キュー (バイナリヒープ) に対するデータ配置方法 N_{struct}, N_{ary} により実装されたダイクストラ法の性能を計算機環境毎 (表 6 参照) にまとめたものである. いずれも関連のある要素をまとめ連続的に配置した (A_{struct}, N_{struct}) を基準 (100%) とした性能比率となっている. ダイクストラ法のように要素間が密接に関連する場合 (同一のタイミング (もしくは短い期間) で要素を要求される) には (A_{struct}, N_{struct}) を選択することが重要である. 計算機環境毎に性能特性は異なるものの性能順は一致していることが確認される.

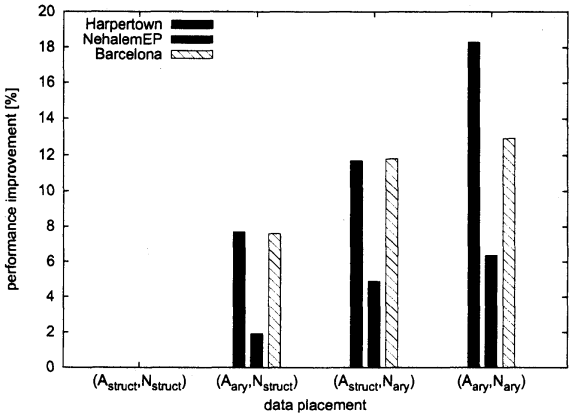


図 9: (A_{struct}, N_{struct}) を基準としたデータ配置毎の性能比率 [%]

表 5: (A_{struct}, N_{struct}) を基準としたデータ配置毎の性能比率 [%]

	(A_{struct}, N_{struct})	(A_{ary}, N_{struct})	(A_{struct}, N_{ary})	(A_{ary}, N_{ary})
Harpertown	0.0%	+7.69%	+11.70%	+18.27%
Nehalem-EP	0.0%	+1.90%	+4.89%	+6.36%
Barcelona	0.0%	+7.59%	+11.80%	+12.92%

表 6: 計算機環境

	プロセッサ	搭載メモリ	OS(Linux)	GCC
Harpertown	Intel Xeon(R) X5460 3.16GHz ×2 (4 コア ×2)	48GB	CentOS 5.4	4.1.2
Nehalem-EP	Intel Xeon(R) X5550 2.67GHz ×2(4 コア ×2)	72GB	Fedora 12	4.4.3
Barcelona	AMD Opteron(tm) 2356 2.30GHz ×2(4 コア ×2)	36GB	CentOS 5.4	4.1.2

2.5 メモリ階層構造上の律速箇所の解析

計算機のメモリ階層構造を考慮した汎用的な評価を行うための実験方法を示す. 本実験で用いる計算機はいずれもクアッドコア・プロセッサを 2 基搭載した計算機環境 (表 6 参照) であるが, Harpertown(図 10 参照) は L2 キャッシュメモリを 2 プロセッサコアで共有しているのに対し, Nehalem-EP(図 11 参照), Barcelona は L2 キャッシュメモリはプロセッサコア毎に配置され新たに 4 プロセッサコア共通の L3 キャッシュメモリが配置されている. なお, コンパイラは GNU GCC(gcc-4.1.2, gcc-4.4.3), 最適化オプションは-O2 を用いる.

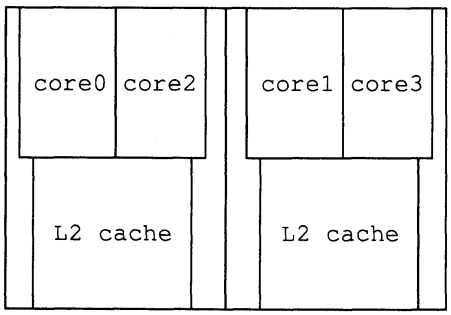


図 10: L2 キャッシュメモリを共有したクアッドコア・プロセッサ

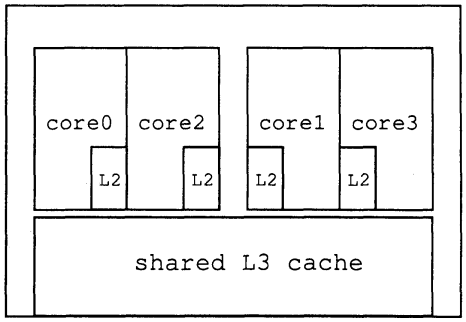


図 11: L2 キャッシュメモリを共有しないクアッドコア・プロセッサ

2.5.1 2 プロセッサコア同時実行

1 プロセッサコア上での実行時間と, 特定の 2 プロセッサコア上で同時実行した際の実行時間を比較することで, メモリ階層構造上の律速箇所を特定することが可能である (表 8 参照). クアッドコア・プロセッサ

を 2 基搭載した計算機環境では, プロセッサコア指定の組合せは表 7 となる. 実行するプロセッサコアの指定には, Linux 上の numactl コマンドを用いる.

表 7: クアッドコア・プロセッサにおける 2 プロセッサコアの組合せ

コアの組合せ	詳細
1 コア	基準とする実行時間
異なるソケット	異なるソケット上の 2 コア
L2 非共有コア	同一ソケット上の L2 キュッシュメモリを共有しない 2 コア (例:図 10 の 0 番と 1 番)
L2 共有コア	同一ソケット上の L2 キュッシュメモリを共有する 2 コア (例:図 10 の 0 番と 2 番)

表 8: 1 プロセス実行性能に対する 2 プロセス同時実行による性能変化 (低下) と律速箇所の関係

律速箇所	異なるソケット	L2 非共有 2 コア	L2 共有 2 コア
メモリ帯域幅 (1 プロセス分)	変化あり	変化あり	変化あり
メモリ帯域幅 (2 プロセス分)	変化なし	変化あり	変化あり
L2 キャッシュメモリの共有	変化なし	変化なし	変化あり
演算性能	変化なし	変化なし	変化なし

まず, Harpertown(表 6, 図 10 参照) において, 優先キュー毎の実験結果をまとめる (表 9 参照). バイナリヒープを適用したダイクストラ法 2-heap は, 他の優先キューと比べて性能低下が最も小さく非常に効率的であることが確認される.

表 9: 2 プロセッサコア同時実行による優先キュー毎の性能比率 [%](実行時間 [sec.])

	1 コア	異なるソケット	L2 非共有 2 コア	L2 共有 2 コア
2-heap	0.00%(5.46)	-1.05%(5.52)	-4.03%(5.69)	-18.56%(6.70)
buckets	0.00%(3.93)	-4.34%(4.13)	-12.48%(4.56)	-30.27%(5.60)
mbp	0.00%(5.69)	-2.00%(5.85)	-6.72%(6.17)	-26.90%(7.73)

次に 2-heap に対する計算機環境毎の実験結果を表 10 にまとめる. 主要な律速箇所はメモリ帯域幅ではなく, L2 キャッシュメモリの共有による性能低下であると確認される. マルチコアプロセッサ環境上で複数のダイクストラ法を並列計算する際に, L2 キャッシュメモリを共有しないようなプロセッサコアに割り当てることで並列効率の高い実行が期待できる.

表 10: 2 プロセッサコア同時実行による計算機環境毎の性能比率 [%](実行時間 [sec.])

	1 コア	異なるソケット	L2 非共有 2 コア	L2 共有 2 コア
Harpertown	0.00%(5.46)	-1.05%(5.52)	-4.03%(5.69)	-18.56%(6.70)
Nehalem-EP	0.00%(5.51)	+0.53%(5.49)	-3.13%(5.69)	-(-)
Barcelona	0.00%(9.80)	+0.02%(9.80)	-5.81%(10.40)	-(-)

2.6 メモリ階層構造による高速化後の性能

9thDIMACS ベンチマーク問題から最大の規模である全米道路ネットワークグラフ (表 11 参照) を用いて性能比較実験を行う.

表 11: 全米道路ネットワークグラフ

変数名	点数 n	枝数 m	枝長 $[u, U]$	次数 $[d, D]$
値	23,947,347	58,333,344	[1, 368,855]	[1, 8]

2.6.1 1 対全最短路問題に対する優先キュー毎のダイクストラ法の性能

全米道路ネットワークグラフ (表 11 参照) に対し SSSP(1 対全最短路問題) の実行時間を示す. 計算機のメモリ階層構造考慮した高速化を行ったバイナリヒープを適用したダイクストラ法 2-heap, 1 レベルバケット

を適用したダイクストラ法 buckets とマルチレベルバケットを適用したダイクストラ法 mbp を表 12 にまとめる. 表中の SSSP-time は始点を 1 とする 1 対全最短路問題の計算時間 (単位は秒), Graph Construction はグラフデータの読み込みとグラフ表現の構築時間 (単位は秒), Memory はメモリ要求量 (単位は GByte) をそれぞれ表している. なお, 計算機環境は Harpertown(表 6 参照) である. 本研究で実装した 2-heap, buckets(表 12 内の太字) はメモリ要求量を抑えながら高速な実行が可能である. また, グラフデータを予めテキストファイルからバイナリファイルに変換しておくことで, 他のソルバーに比べ極めて高速な初期化時間での実行を可能となる. バイナリデータの作成に要する時間は数十秒である.

表 12: 性能比較実験

	計算量	アルゴリズム	SSSP-time[sec.]	Graph Construction[sec.] text-file	Graph Construction[sec.] binary-file	Memory[GB]
2-heap	$O(m \log_2 n)$	binary heap	**5.53**	**30.30**	**0.91**	**0.90**
buckets	$O(m + nC)$	Dial's algorithm	**3.50**	**30.30**	**0.91**	**0.91**
mbp[6]	$O(m + n \log C)$	multi-level buckets	5.68	51.70	-	2.17

2.6.2 1 対 1 最短路問題に対する計算機環境毎のバイナリヒープを適用したダイクストラ法の性能

表 13, 図 12 は, 2-heap の計算機環境毎 (表 6 参照) のクエリ単位の実行時間をまとめたものである. いずれの計算機環境においても搭載コア数すべてを使用した 8 スレッド並列時の実行時間が最も短い. L2 キャッシュメモリを 2 プロセッサコアで共有するクアッドコア・プロセッサ (Harpertown) では, 8 スレッド時 (L2 キャッシュメモリを共有するコアにも割り当てられる) に並列効率が低下していることが確認され, 表 10 で得られた結果と合致する. 一方, プロセッサコア毎に L2 キャッシュメモリを保持しているクアッドコア・プロセッサ (NehalemEP, Barcelona) では並列効率の低下は非常に小さい.

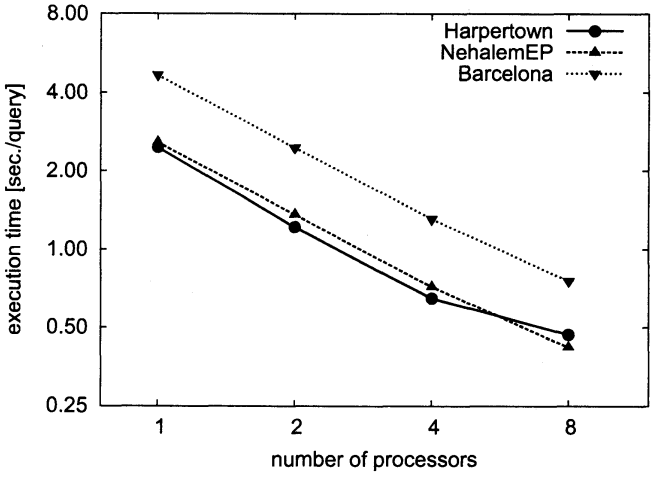


図 12: 計算機環境毎のダイクストラ法の実行時間 [sec./query]

表 13: 計算機環境毎のダイクストラ法の実行時間 [sec./query]

number of processors	1	2	4	8
Harpertown	2.47	1.22	0.65	0.47
NehalemEP	2.58	1.36	0.72	0.42
Barcelona	4.67	2.46	1.31	0.76

2.6.3 1 対 1 最短路問題に対する優先キュー毎のダイクストラ法の性能

図 13, 14, 表 14,15 は, 計算機環境 Harpertown(表 6 参照) 上での 2-heap, buckets, mbp のクエリ単位の実行時間をまとめたものである. スレッド並列計算に未対応の mbp に比べ, 2-heap, buckets はスレッド並

列計算により非常に高速に計算することが可能である. 特に 2-heap は 4 スレッド並列実行時に mbp と比べて同量のメモリ要求量で 4.02 倍高速である (表 14, 15 内の太字).

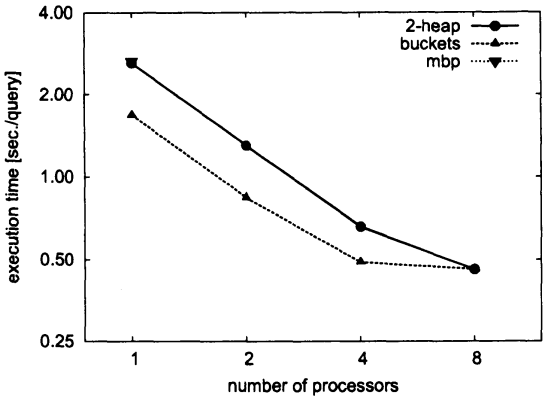


図 13: 優先キュー毎の実行時間 [sec./query]

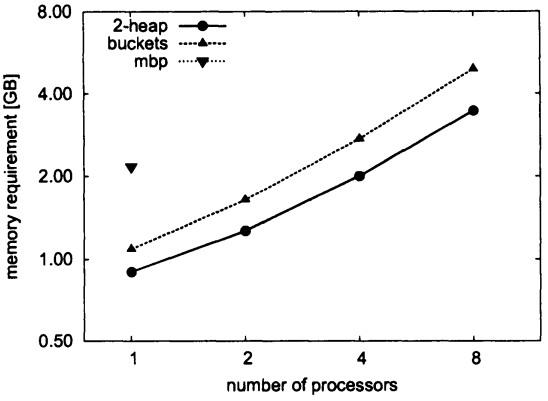


図 14: 優先キュー毎のメモリ要求量 [GB]

表 14: 優先キュー毎の実行時間 [sec./query]

number of processors	1	2	4	8
2-heap	2.61	1.30	**0.66**	0.46
buckets	1.68	0.84	0.49	0.46
mbp	**2.65**	-	-	-

表 15: 優先キュー毎のメモリ要求量 [GB]

number of processors	1	2	4	8
2-heap	0.90	1.27	**2.00**	3.46
buckets	1.09	1.64	2.73	4.93
mbp	**2.17**	-	-	-

3 時空間ネットワークに対する経路検索

首都圏鉄道ネットワークを時間軸方向に拡張した時空間ネットワークを構築し電車毎の運行を表現することで, 鉄道利用者の動的な流れを解析することが可能である [15, 16, 17]. 時空間ネットワークの構築には電車の発着時刻が確定的であることを利用し, 各電車の発着ごとに点 (停車ノード) を作成し, それらを電車の駅間移動を表す枝 (走行リンク, 着発間リンク, 待ちリンク, 待ち合わせリンク, 乗り換えリンク) で結ぶことにより, 乗客の動的な流れを静的なネットワークの流れとして表現する. 時空間ネットワークを用いることで大規模化してしまうものの, 時間軸方向の近似や乗客の集約をすることなく, 乗客の移動を正確に表現することが可能となる. また動的な定式化に比べ扱いが容易である.

対象とする鉄道ネットワークは, 2000 年に実施された大都市交通センサスを対象とした首都圏 128 路線, 1815 駅である (図 15 参照). 電車の発着時刻は, 市販の時刻表の電子データである「全国 JR 時刻表 2005 年 1 月号」と「JTB パブリッシング私鉄データ (2005 年 2 月 10 日現在)」より取得した.

表 16: 首都圏鉄道ネットワークグラフの規模

変数名	規模	説明
点数	783,533	停車ノード
枝数	3,113,843	走行リンク, 着発間リンク, 待ちリンク, 待ち合わせリンク, 乗り換えリンク
停車ノード	783,533	各駅における各電車の停車を表現
走行リンク	530,589	電車に乗り次の駅への移動を表現
着発間リンク	218,489	駅での停車を表現
待ちリンク	558,012	次の電車を待つことを表現
待ち合わせリンク	21,084	電車の待ち合わせを表現
乗り換えリンク	1,785,669	別の路線に乗り換えを表現

3.1 時空間ネットワークに対する経路検索

時空間ネットワーク (表 16,17 参照) に対して, 計算機のメモリ階層構造考慮した高速化を行ったバイナリヒープを適用したダイクストラ法 2-heap, 1 レベルバケットを適用したダイクストラ法 buckets を用いた経

図 15: 大都市センサスの対象範囲

図 16: 時空間ネットワーク

路探索の結果を表 18 にまとめる. Harpertown 計算機環境 (表 6 参照) 上で, 8 スレッド並列計算を行った. 表中の APSP-time は 全対全最短路問題の計算時間 (単位は秒), 括弧内の SSSP-time は SSSP あたりの計算時間 (単位はミリ秒), Graph Construction はグラフデータの読み込みとグラフ表現の構築時間 (単位は秒), Memory はメモリ要求量 (単位は GByte) をそれぞれ表している. 最大枝長が小さいため, バケットを用いた優先キューとの相性が良く, buckets の実行時間が 2-heap に比べ短い. APSP の計算時間は buckets では約 2.98 時間, 2-heap では約 3.65 時間である.

表 17: 首都圏鉄道ネットワークグラフ

変数名	点数 n	枝数 m	枝長 $[u, U]$	次数 $[d, D]$
値	783,533	3,113,843	[0, 1189]	[0, 37]

表 18: 時空間ネットワーク上の経路探索

	APSP-time[sec.](SSSP-time[msec.])	Graph Construction[sec.]	Memory[MB]
2-heap(8threads)	13132.483(16.761)	1.519	123.14
buckets(8threads)	10721.697(13.684)	1.519	170.25

4 おわりに

計算機のメモリ階層構造を考慮することにより, 大規模最短路問題に対するダイクストラ法に対して汎用的かつ大幅な高速化を行うことが可能であることを示した. 本研究で高速化を施したバイナリヒープを適用したダイクストラ法は, 先行研究のマルチレベル・バケットに対して 1 スレッド時には同程度の性能を示し, メモリ要求量が同量となる 4 スレッド時には 4.02 倍高速である. 数値実験により本ソフトウェアは L2 キャッシュメモリ帯域幅に律速しているが確認され, L2 キャッシュメモリを共有しないプロセッサコアに割り当てることで, 非常に効率的なマルチスレッド計算が可能である. グラフ特性に対するメモリ要求量や実行時間の安定性, 省メモリ性, マルチコア・プロセッサ環境での性能など, 総合的に評価すると最も優れているといえる. 本研究で扱った実装方法, 評価手法は非常に汎用的であり, ダイクストラ法や優先キューだけに限定せずにアルゴリズム実装に広く適用可能で高速化に期待ができる.

また時空間ネットワークグラフに対しても, 1 台の計算機上で全対全最短路問題を数時間 (buckets では 2.98 時間, 2-heap では 3.65 時間) で計算が可能である. 今後の展開として, 時空間ネットワークの対象を首都圏鉄道ネットワークから首都圏規模の道路ネットワーク, 歩行者ネットワークへと拡張し, 大規模災害時を想定した避難経路探索, 交通管制を考えている.

参考文献

[1] E. W. Dijkstra: A Note on Two Problems in Connexion with Graphs. Numerische Mathematik, 1:269.271. 1959.

[2] J.W. J.Williams: Heapsort. Communications of the ACM, 7:347.348. 1964.

[3] R. B. Dial: Algorithm 360: Shortest Path Forest with Topological Ordering. Comm. ACM, 12:632-633 1969.

[4] M. L. Fredman and R. E. Tarjan: Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. J. Assoc. Comput. Mach., 34:596-615. 1987.

[5] B. V. Cherkassky, Andrew V. Goldberg, T Radzik: Shortest paths algorithms: theory and experimental evaluation. Mathematical programming. 1996.

[6] Andrew V. Goldberg: A Simple Shortest Path Algorithm with Linear Average Time. Technical Report STAR-TR-01-03, STAR Lab., InterTrust Tech., Inc., Santa Clara, CA, USA. 2001.

[7] 9th DIMACS Implementation Challenge. `http://www.dis.uniroma1.it/~challenge9/`.

[8] Camil Demetrescu, Andrew V. Goldberg, David S. Johnson: 9th DIMACS Implementation Challenge Core Problem Families. 2006.

[9] Camil Demetrescu, Andrew V. Goldberg, David S. Johnson: 9th DIMACS Implementation Challenge Benchmark Solvers. 2006.

[10] GotoBLAS. `http://www.tacc.utexas.edu/resources/software/`.

[11] Kazushige Goto, K. and Robert A. van de Geijn: On reducing TLB misses in matrix multiplication. Tech.Rep. CS-TR-02-55. 2002.

[12] Kazushige Goto, K. and Robert A. van de Geijn: High-performance implementation of the level-3 BLAS. FLAME Working Note #20 TR-2006-23. 2006.

[13] John L. Hennessy and David A. Patterson: Computer Architecture: A Quantitative Approach (Third Edition ed.). Morgan Kaufmann Publishers.

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C.Stein: Introduction to Algorithms. MIT Press, 2nd edition. 2001.

[15] 田口東: 首都圏電車ネットワークに対する時間依存通勤交通配分モデル. 日本オペレーションズ・リサーチ学会和文論文誌 Vol.48 pp.85-108. 2005 .

[16] 鳥海重喜, 中村幸史, 田口東: 通勤電車の遅延計算モデル. オペレーションズ・リサーチ Vol.50 No.6 pp.409-416. 2005.

[17] 鳥海重喜, 川口真由, 田口東: 首都直下地震による鉄道利用通勤・通学客の被害想定. オペレーションズ・リサーチ Vol.53 No.2 pp.111-118. 2008

[18] 安井雄一郎, 藤澤克樹, 笹島啓史, 後藤和茂: 大規模最短路問題に対するダイクストラ法の高速化. 日本オペレーションズ・リサーチ学会和文論文誌. 掲載予定.